

Nro. ord.	Apellido y nombre	L.U.	Turno	#hojas
				4

SISTEMAS DIGITALES - Segundo Parcial
Segundo Cuatrimestre 2024

Ej. 1	Ej. 2	Ej. 3	Ej. 4	Nota
2	2	4	2	10

Correctorx:
M. J. 100

Aclaraciones

- Anote apellido, nombre, LU y numere *todas* las hojas entregadas, entregando los distintos ejercicios en hojas separadas.
- El parcial **no es a libro abierto** pero pueden consultar la hoja de referencia provista por la cátedra.
- Justifique sus respuestas explicando lo que considere necesario en lenguaje natural.
- El parcial se aprueba con 6 y se deben tener ambos parciales aprobados para aprobar la materia (promoción directa).

Ejercicio 1 (2 pts.) Implemente la función que calcula el valor de un elemento en un triángulo de Pascal, definido en el siguiente bloque de código C.

```
function pascal(fila, columna) {
    if (columna > fila) return 0;
    if (columna <= 1 || fila <= 1) return 1;
    return pascal(fila - 1, columna) + pascal(fila - 1, columna - 1);
}
```

El triángulo de Pascal define una secuencia triangular de números enteros que comienza con un 1 (uno) en el vértice superior y se expande hacia abajo con números calculados a partir de los números de la fila superior. Cada número en el triángulo es la suma de los dos números directamente arriba de él.

Más allá de la interpretación de la función les recomendamos concentrarse en la traducción a código RISC V que respete la convención de llamada. Deben explicar en qué registros se almacenan los valores en cada paso y cómo se aseguran que se respeta la convención.

Ejercicio 2 (2 pts.) Implemente las siguientes funciones en lenguaje ensamblador de RISC V respetando la convención de llamada presentada en la materia. Describir el comportamiento y cómo se aseguran que se respete la convención.

- **int inv(int x) = -x** (inverso aditivo)
- **void invertirArreglo(int arr[], int largo)**: Dado un puntero a un arreglo de enteros de 32 bits y la cantidad de elementos, cambia cada valor del arreglo por su inverso aditivo.

Ejercicio 3 (4 pts.) Se tiene una estructura **BalanceDeudor** que contiene el ID del cliente como un entero sin signo de 8 bits, la suma de sus consumos como un entero en complemento a dos de 16 bits y la suma de sus pagos realizados como un entero en complemento a dos de 16 bits. Ubicación de los datos de una estructura **BalanceDeudor**:

Byte	0x0000	0x0001	0x0003
Nombre	ID	Consumos	Pagos

En memoria se encuentra un arreglo **balanceDeudores** del tipo **BalanceDeudor** con la forma:

Direccion	0x0000	0x0001	0x0003	0x0005	...	0x0030	0x0031	0x0033	0x0035
Valor	17	30020	1232	6	...	9	5878	300	0

Donde el final del arreglo es demarcado por un ID nulo. Se pide

- Calcular cuántos bytes ocupa en memoria la estructura **BalanceDeudor** y cuántos bytes un arreglo de tipo **BalanceDeudor** de doce elementos.
- Escribir una función **contarDeudores(balanceDeudores)** que dada una posición de memoria que indica el comienzo de un arreglo **balanceDeudores** de **BalanceDeudor**, devuelva la suma estructuras donde la suma de los consumos (segundo elemento de la estructura) es mayor que la suma de los pagos realizados (tercer elemento de la estructura). Ejemplo:

```

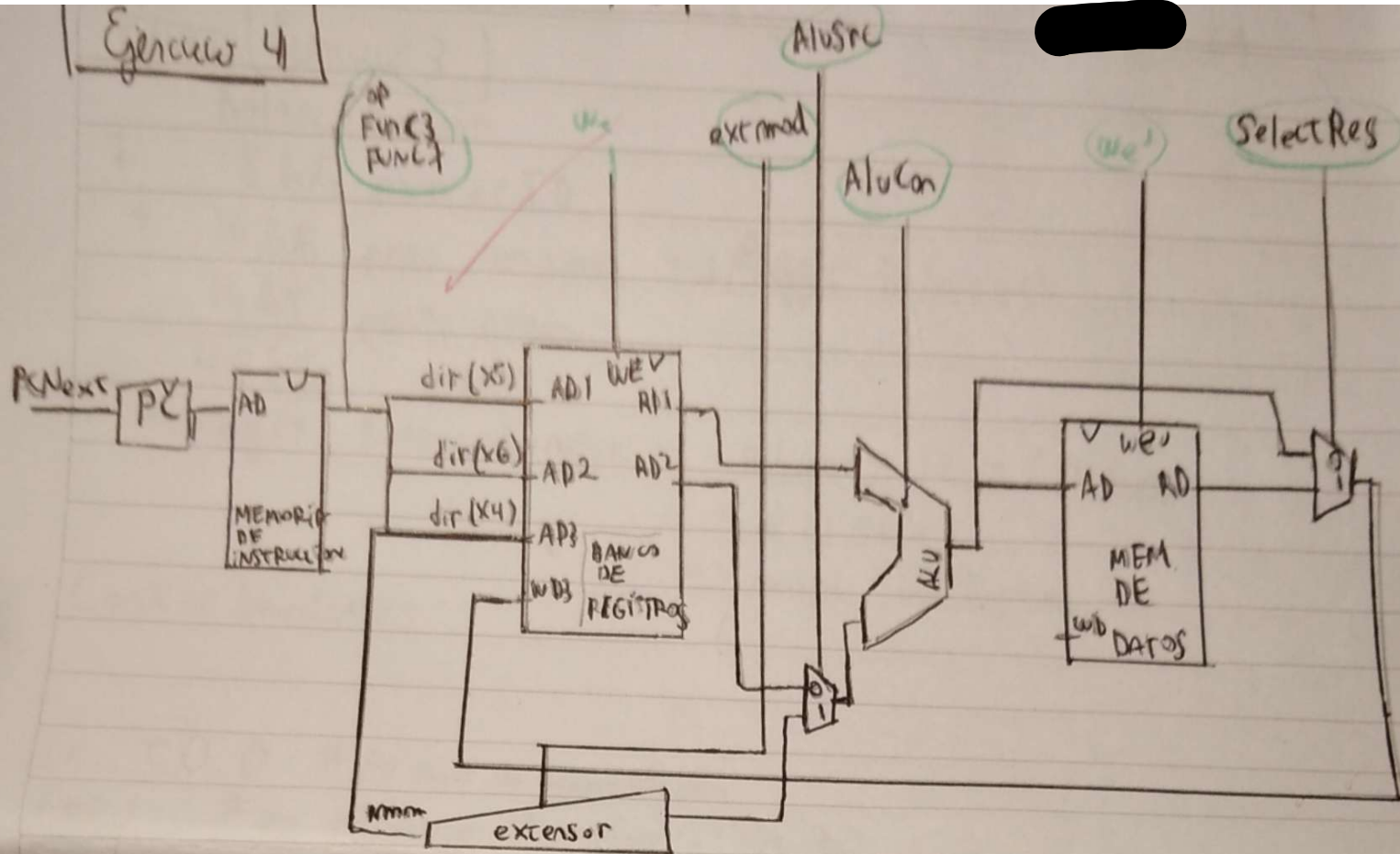
.data
balanceDeudores: .byte 17
                  .half 30020
                  .half 1232
                  .byte 6
                  .half 200
                  .half 200
                  .byte 9
                  .half 5878
                  .half 300
                  .half 0      #Declaramos el final del arreglo

.text
contarDeudores:
    ...

```

Para este caso `balanceDeudores` debe devolver 2 ya que el usuario con id 17 y el usuario con id 9 tienen consumos mayores a sus pagos. Recuerden que el arreglo es pasado como dirección de memoria de su primer elemento a través del primer parámetro de la función.

Ejercicio 4 (2 pts.) Para una microarquitectura de ciclo simple para un procesador de RISC V, como la que vimos en clase, explicar que componentes y señales de control están involucrados y cómo modifican el estado del procesador al ejecutar la instrucción `or x4, x5, x6`.



~~MEM~~ or x4, x5, x6

- Primero se carga la instrucción "or x4, x5, x6" desde la memoria de instrucción
 - De ahí se "parte" la instrucción: OP, FUNC3 y FUNC7 van a la unidad de control y la dirección de x5, x6 y x4 van a AD1, AD2 y AD3 respectivamente. ~~El inmediato va al extensor~~
 - El valor de x5 sale por RD1 y el de x6 por RD2. De ahí ~~el valor~~ de RD2 va a un multiplexor (controlado por la unidad de control). La señal AluSrc tiene que estar en 0
 - Luego pasa por la ALU. De donde la unidad de control tiene que haber mandado por AluCon el valor correspondiente a un or
 - La salida de la ALU pasa por un multiplexor; controlado por la unidad de control. Nota que SelectRes debe estar en 0
 - La salida de dicho multiplexor va a WB3, con WE alta para que se escriba el resultado en la dirección que entra al banco de registros por WB3
- Aclaración: Del esquema de arriba omití el mecanismo para actualizar el PC ya que no lo considero tan relevante para lo que pide el ejercicio (hay otra aclaración en la siguiente carilla)

Aclaración 2:

Lo que marqué en verde va a la unidad de control

Ejercicio 3

Balance Deudor

- 8 bits para el ID
- + 16 bits para consumo \Rightarrow 5 bytes
- 16 bits para pago
- 40 bits

- Arreglo Balance Deudor: 5 bytes $\cdot 12 = 60$ bytes
+ el ID mudo
 \Rightarrow ocupa 61 bytes

Contar Deudores:

li t0, 0 # lo uso de acumulador

for: # en a0 tengo el puntero al arreglo

lbu t4, 0(a0) # cargo el ID en t4

bge t4, return # por

lh t1, 1(a0) # cargo el consumo en t1

Ma

lh t2, 3(a0) # cargo el pago en t2

sub t3, t2, t1 # t3 \leftarrow PAGO - CONSUMO

bge t ~~bge~~ t3, no_hacer_nada

addi t0, t0, 1 # incremento mi acumulador

no_hacer_nada:

addi a0, a0, 5 # para ir al siguiente elemento

j for

return:

mv a0, t0

ret

Ejercicio 2

inv:

xori a0, a0, -1 # doy vuelta los bits de a0

addi a0, a0, 1

ret

no tocar ninguno de los registros protegidos y estoy terminando

mi único argumento por a0. Por tanto estoy respetando la convención

Invertir Arreglo: # a0 = puntero a arreglo, a1 = cont. de elementos

addi sp, sp, -16 # reservo memoria en el stack

sw ra, 0(sp) # guardo r2 (para poder respetar la convención)

sw s1, 4(sp) # para poder restaurarlos más tarde y respetar la

sw s2, 8(sp) # convención

sw s3, 12(sp)

~~lw t0, 0(a0) # cargo el elemento al que apunta a0 en t0~~

mv s1, a0 # Según la interfaz lineal de aplicación luego de la llamada a

mv s2, a1 # inv no tengo garantía de que [a0, a1] se mantengan.

li s3, 0 # lo uso de índice

For:

beg s3, s2, return

lw t0, 0(s1)

mv a0, t0

jal inv

sw a0, 0(s1)

addi s3, s3, 1

addi s1, s1, 4

j for

El return está en la siguiente casilla

return:

```
lw ra, 0(sp) # Restaura los registros protegidos antes de salir de la función
lw s1, 4(sp)
lw s2, 8(sp)
lw s3, 12(sp)
addi sp, sp, 16
ret
```

*1: Según la convención los registros $ra, [s0, s11]$ ~~son protegidos~~ son protegidos. Esto implica que la función llamadora puede suponer que luego de la llamada a la función llamada van a tener el mismo valor que antes de entrar a dicha función.

*2: Esto lo hice porque más abajo en el código hago una llamada a inv y por la razón mencionada en *1.

Ejercicio 1
Ejercicio 1

pascal: # a0 = Fila, a1 = columna

addi sp, sp, -32

sw s1, 0(sp)

sw s2, 4(sp)

sw s3, 8(sp)

sw s4, 12(sp)

sw ra, 16(sp)

bgt a1, a0, return_0 # if (columna > Fila) return 0

li t0, 1

ble a1, t0, return_1 # if (columna <= 1 || Fila <= 1) return 1;

ble a1, t0, return_1 #

caso de caminos

mv s1, a0 # save copia a0 a S1

mv s2, a1 # " a1 a S2

addi a0, a0, -1

jal pascal # pascal(Fila-1, columna)

mv s3, a0 # guarda el resultado en S3

~~return~~ addi a0, s1, -1

~~return~~ addi a1, s2, -1

jal pascal # pascal(Fila-1, columna-1)

add a0, a0, s3 # a0 = pascal(Fila-1, columna) + pascal(Fila-1, columna-1)

return

Los return están en la otra celda


```
return_0: li a0, 0
j return
return_1: li a0, 1
j return
```

```
return:
lw s1, 0(sp)
lw s2, 4(sp)
lw s3, 8(sp)
lw s4, 12(sp)
lw ra, 16(sp)
addi sp, sp, 32
ret
```